Leibniz-Rechenzentrum
der Bayerischen Akademie der Wissenschaften

Performance Optimization of Smoothed Particle Hydrodynamics and Experiences on Many-Core Architectures

Dr. Luigi Iapichino

luigi.iapichino@lrz.de

Dr. Fabio Baruffa

Leibniz Supercomputing Centre

Intel MIC Programming Workshop & Scientific Workshop "HPC for natural hazard assessment and disaster mitigation", LRZ, June 28th, 2017

Work contributors

Dr. Fabio Baruffa
Sr. HPC Application Specialist
Leibniz Supercomputing Centre

- Member of the Intel Parallel Computing Center (IPCC) @ LRZ/TUM
- Expert in performance optimization and HPC systems

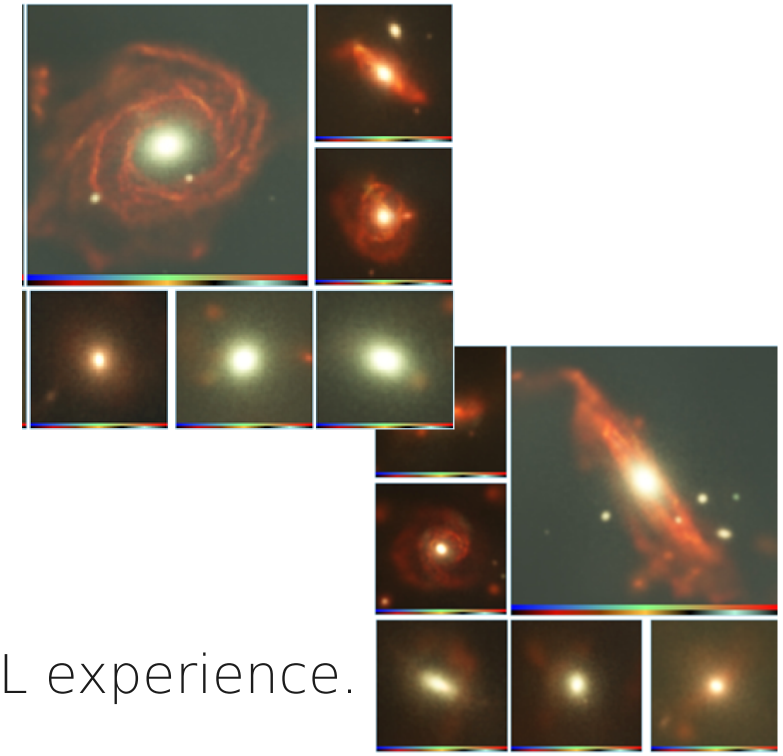
Dr. Luigi Iapichino
Scientific Computing Expert
Leibniz Supercomputing Centre

- Member of the Intel Parallel Computing Center (IPCC) @ LRZ/TUM
- Expert in computational astrophysics and simulations

Email contacts: fabio.baruffa@lrz.de luigi.iapichino@lrz.de

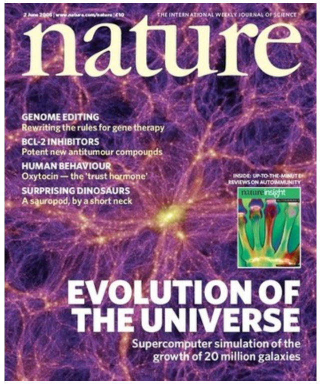
Outline of the talk

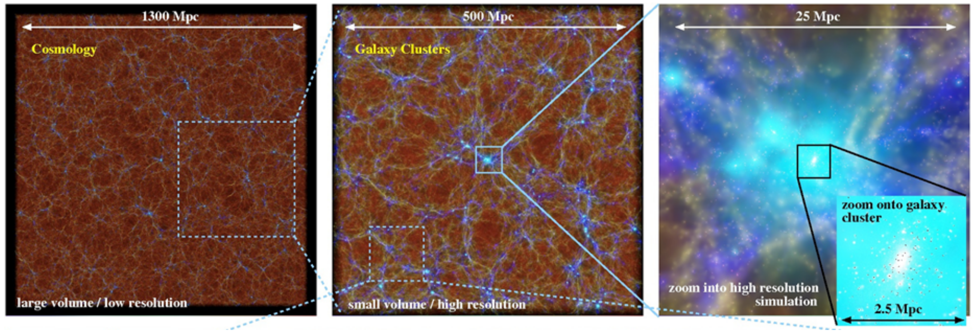
- Overview of the code: P-Gadget3 and SPH.
- Challenges in code modernization approach.
- Multi-threading parallelism and scalability.
- Enabling vectorization through:
  - Data layout optimization (AoS → SoA).
  - Reducing conditional branching.
- Performance results, takeaways from our KNL experience.



Gadget intro

- Leading application for simulating the formation of the cosmological large-scale structure (galaxies and clusters) and of processes at sub-resolution scale (e.g. star formation, metal enrichment).
- Publicly available, cosmological TreePM N-body + SPH code.
- First developed in the late 90s as serial code, later evolved as an MPI and a hybrid code.
- Good scaling performance up to O(100k) Xeon cores (SuperMUC@LRZ).

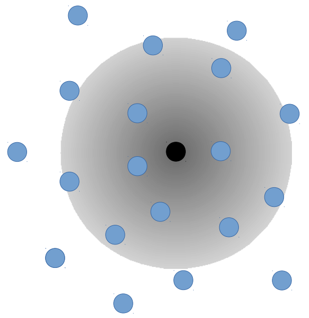

nature
GENOME EDITING
Rewriting the rules for gene therapy
BCL-2 INHIBITORS
Potent new antitumour compounds
HUMAN BEHAVIOUR
Oxytocin — the 'trust hormone'
SURPRISING DINOSAURS
A sauropod, by a short neck
EVOLUTION OF THE UNIVERSE
Supercomputer simulation of the growth of 20 million galaxies


1300 Mpc
Cosmology
large volume / low resolution
500 Mpc
Galaxy Clusters
small volume / high resolution
25 Mpc
zoom onto galaxy cluster
zoom into high resolution simulation
2.5 Mpc

Smoothed particle hydrodynamics (SPH)

- SPH is a Lagrangian particle method for solving the equations of fluid dynamics, widely used in astrophysics.
- It is a mesh-free method, based on a particle discretization of the medium.
- The local estimation of gas density (and all other derivation of the governing equations) is based on a kernel-weighted summation over neighbor particles:

$$\rho_i = \rho(\mathbf{r}_i) = \sum_j m_j W(|\mathbf{r}_i - \mathbf{r}_j|, h_j)$$



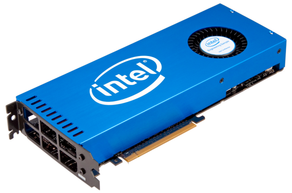
Optimization strategy

- We isolate the representative code kernel subfind_density and run it in as a stand-alone application, avoiding the overhead from the whole simulation.
- As most code components, it consists of two sub-phases of nearly equal execution time (40 to 45% for each of them), namely the neighbour-finding phase and the remaining physics computations.
- Our physics workload: $\sim$ 500k particles. This is a typical workload per node of simulations with moderate resolution.
- We focus on node-level performance, through minimally invasive changes.
- We use tools from the Intel® Parallel Studio XE (VTune Amplifier and Advisor).

Target architectures for our project

Intel® Xeon processor

- E5-2650v2 Ivy-Bridge (IVB) @ 2.6 GHz, 8-cores / socket.
  TDP: 95W, RCP (03/2017): $1116.
- AVX.



Intel® Xeon Phi™ coprocessor
1st generation

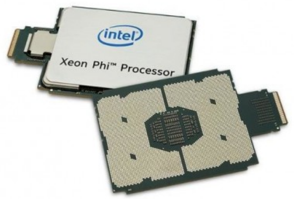
- Knights Corner (KNC) coprocessor 5110P @ 1.1GHz, 60 cores.
  TDP: 225W, RCP: N/D.
- Native / offload computing.
- Directly login via ssh.
- SIMD 512 bits.

Further tested architectures

Intel® Xeon processors

- E5-2697v3 Haswell (HSW) @ 2.3 GHz, 14-cores / socket.
  TDP: 145W, RCP (03/2017): $2702.
- AVX2, FMA.

- E5-2699v4 Broadwell (BDW) @ 2.2 GHz, 22-cores / socket.
  TDP: 145W, RCP (03/2017): $4115.
- AVX2, FMA.



Intel® Xeon Phi™ processor
2nd generation

- Knights Landing (KNL) Processor 7250 @ 1.4 GHz, 68 cores.
  TDP: 215W, RCP (03/2017): $4876.

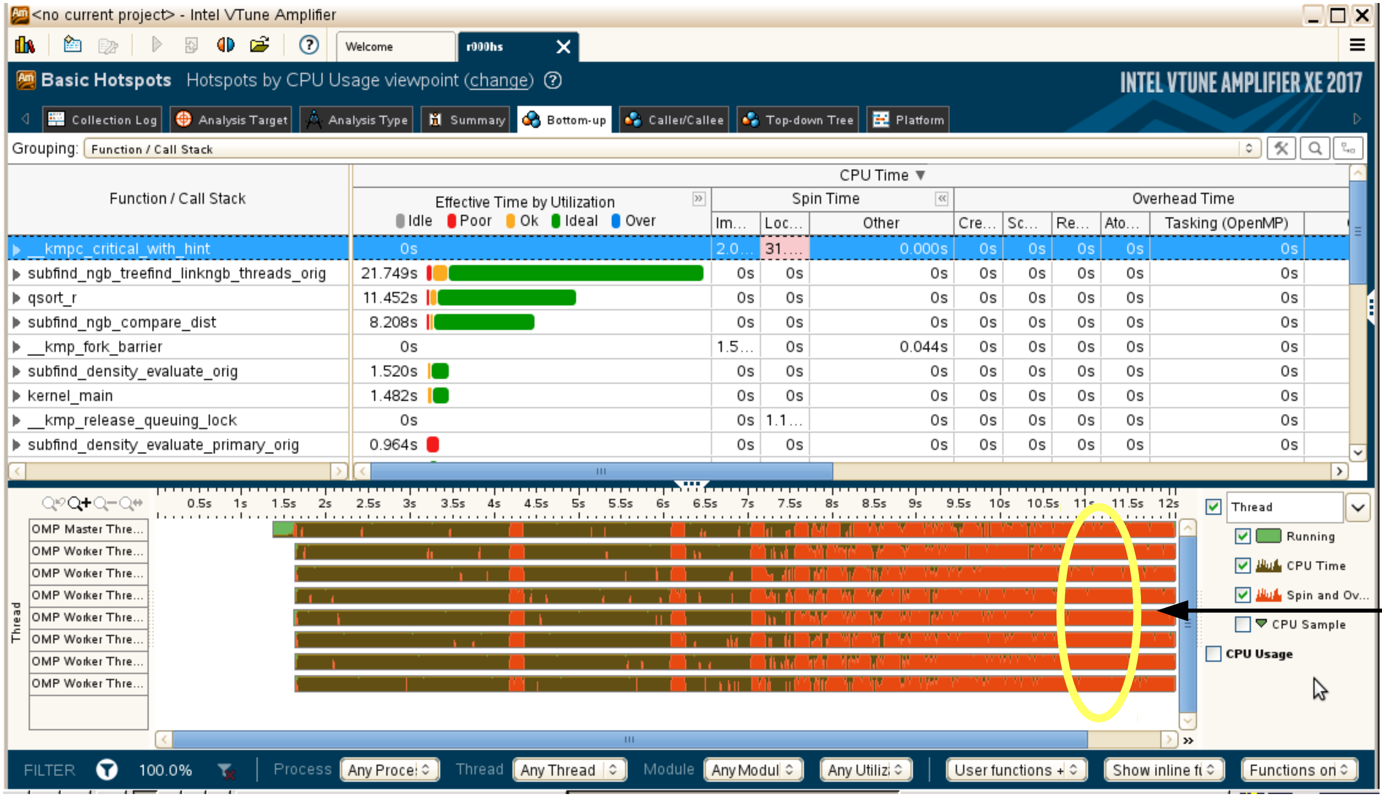
- Available as bootable processor.

- Binary-compatible with x86.

- High bandwidth memory.

- New AVX512 instructions set.

Initial profiling


<no current project> - Intel VTune Amplifier

Welcome | r000hs

Basic Hotspots Hotspots by CPU Usage viewpoint (change)

INTEL VTUNE AMPLIFIER XE 2017

Collection Log | Analysis Target | Analysis Type | Summary | Bottom-up | Caller/Callee | Top-down Tree | Platform

Grouping: Function / Call Stack

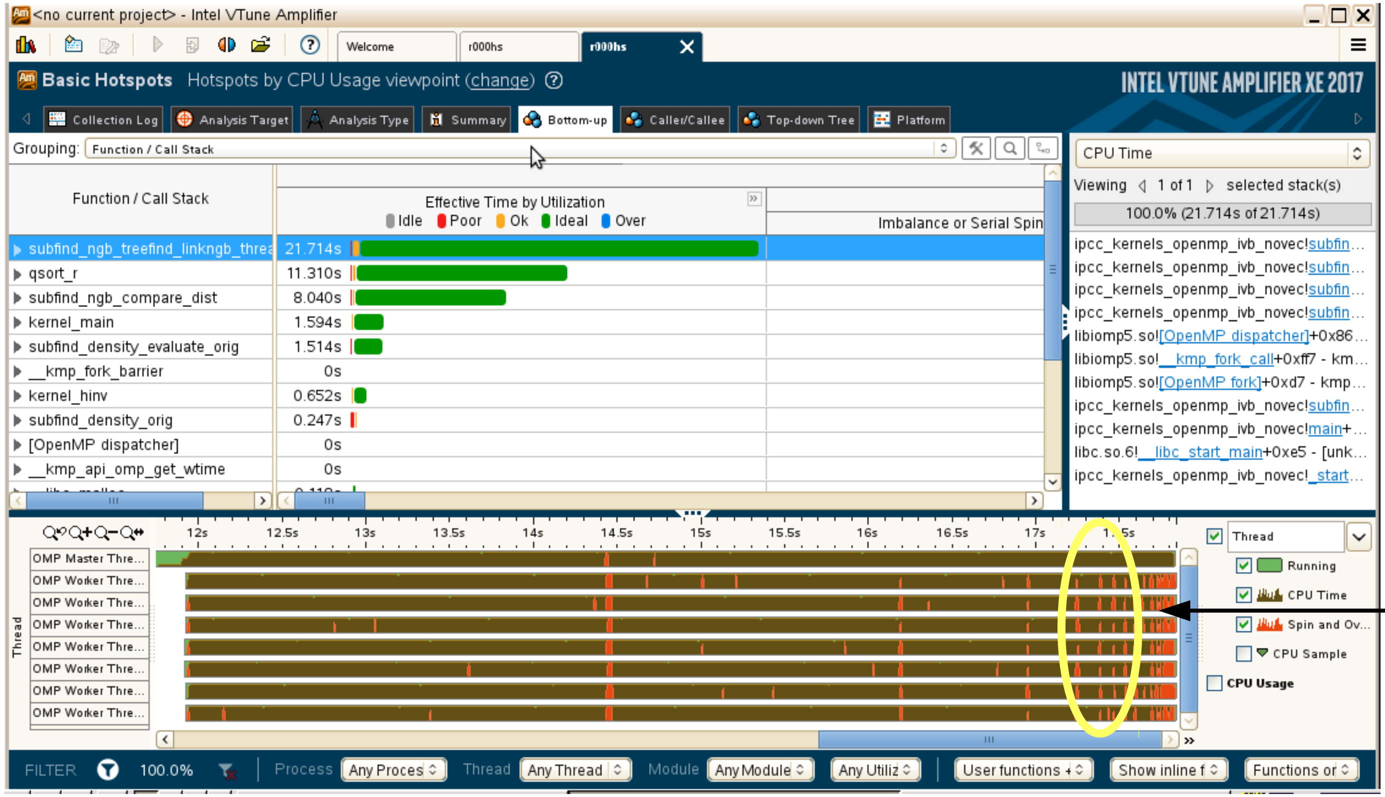
Function / Call Stack	CPU Time: Effective Time by Utilization (Idle, Poor, Ok, Ideal, Over)	Spin Time: Im...	Spin Time: Loc...	Spin Time: Other	Overhead Time: Cre...	Overhead Time: Sc...	Overhead Time: Re...	Overhead Time: Ato...	Overhead Time: Tasking (OpenMP)
__kmpc_critical_with_hint	0s	2.0...	31....	0.000s	0s	0s	0s	0s	0s
subfind_ngb_treefind_linkngb_threads_orig	21.749s	0s	0s	0s	0s	0s	0s	0s	0s
qsort_r	11.452s	0s	0s	0s	0s	0s	0s	0s	0s
subfind_ngb_compare_dist	8.208s	0s	0s	0s	0s	0s	0s	0s	0s
__kmp_fork_barrier	0s	1.5...	0s	0.044s	0s	0s	0s	0s	0s
subfind_density_evaluate_orig	1.520s	0s	0s	0s	0s	0s	0s	0s	0s
kernel_main	1.482s	0s	0s	0s	0s	0s	0s	0s	0s
__kmp_release_queuing_lock	0s	0s	1.1...	0s	0s	0s	0s	0s	0s
subfind_density_evaluate_primary_orig	0.964s	0s	0s	0s	0s	0s	0s	0s	0s

FILTER 100.0% Process Any Proce: Thread Any Thread Module Any Modul Any Utiliz: User functions + Show inline fi Functions on

- Severe shared-memory parallelization overhead
- At later iterations, the particle list is locked and unlocked constantly due to the recomputation
- Spinning time 41%

thread spinning

Improved performance



<no current project> - Intel VTune Amplifier
Welcome
r000hs
r000hs
Basic Hotspots Hotspots by CPU Usage viewpoint (change)
INTEL VTUNE AMPLIFIER XE 2017
Collection Log
Analysis Target
Analysis Type
Summary
Bottom-up
Caller/Callee
Top-down Tree
Platform

Grouping: Function / Call Stack

Function / Call Stack	Effective Time by Utilization (▮Idle 🟥Poor 🟧Ok 🟩Ideal 🟦Over)	Imbalance or Serial Spin
subfind_ngb_treefind_linkngb_threa	21.714s	
qsort_r	11.310s	
subfind_ngb_compare_dist	8.040s	
kernel_main	1.594s	
subfind_density_evaluate_orig	1.514s	
__kmp_fork_barrier	0s	
kernel_hinv	0.652s	
subfind_density_orig	0.247s	
[OpenMP dispatcher]	0s	
__kmp_api_omp_get_wtime	0s	

CPU Time

Viewing 1 of 1 selected stack(s)

100.0% (21.714s of 21.714s)

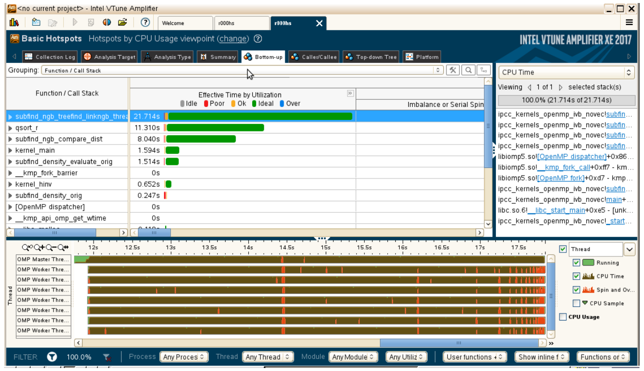
```
ipcc_kernels_openmp_ivb_novec!subfin…
ipcc_kernels_openmp_ivb_novec!subfin…
ipcc_kernels_openmp_ivb_novec!subfin…
ipcc_kernels_openmp_ivb_novec!subfin…
libiomp5.so![OpenMP dispatcher]+0x86…
libiomp5.so!__kmp_fork_call+0xff7 - km…
libiomp5.so![OpenMP fork]+0xd7 - kmp…
ipcc_kernels_openmp_ivb_novec!subfin…
ipcc_kernels_openmp_ivb_novec!main+…
libc.so.6!__libc_start_main+0xe5 - [unk…
ipcc_kernels_openmp_ivb_novec!_start…
```

FILTER 100.0% Process Any Proces Thread Any Thread Module Any Module Any Utiliz User functions + Show inline f Functions or

- Lockless scheme: lock contention removed through "todo" particle list and OpenMP dynamic scheduling.
- Time spent in spinning only 3%

no spinning

Improved speed-up


<no current project> - Intel VTune Amplifier

Basic Hotspots Hotspots by CPU Usage viewpoint (change)

Grouping: Function / Call Stack

Function / Call Stack	Effective Time by Utilization	Imbalance or Serial Spin
subfind_ngb_treefind_linkngb_threa	21.714s	
qsort_r	11.310s	
subfind_ngb_compare_dist	8.040s	
kernel_main	1.594s	
subfind_density_evaluate_orig	1.514s	
__kmp_fork_barrier	0s	
kernel_hinv	0.652s	
subfind_density_orig	0.247s	
[OpenMP dispatcher]	0s	
__kmp_api_omp_get_wtime	0s	

CPU Time

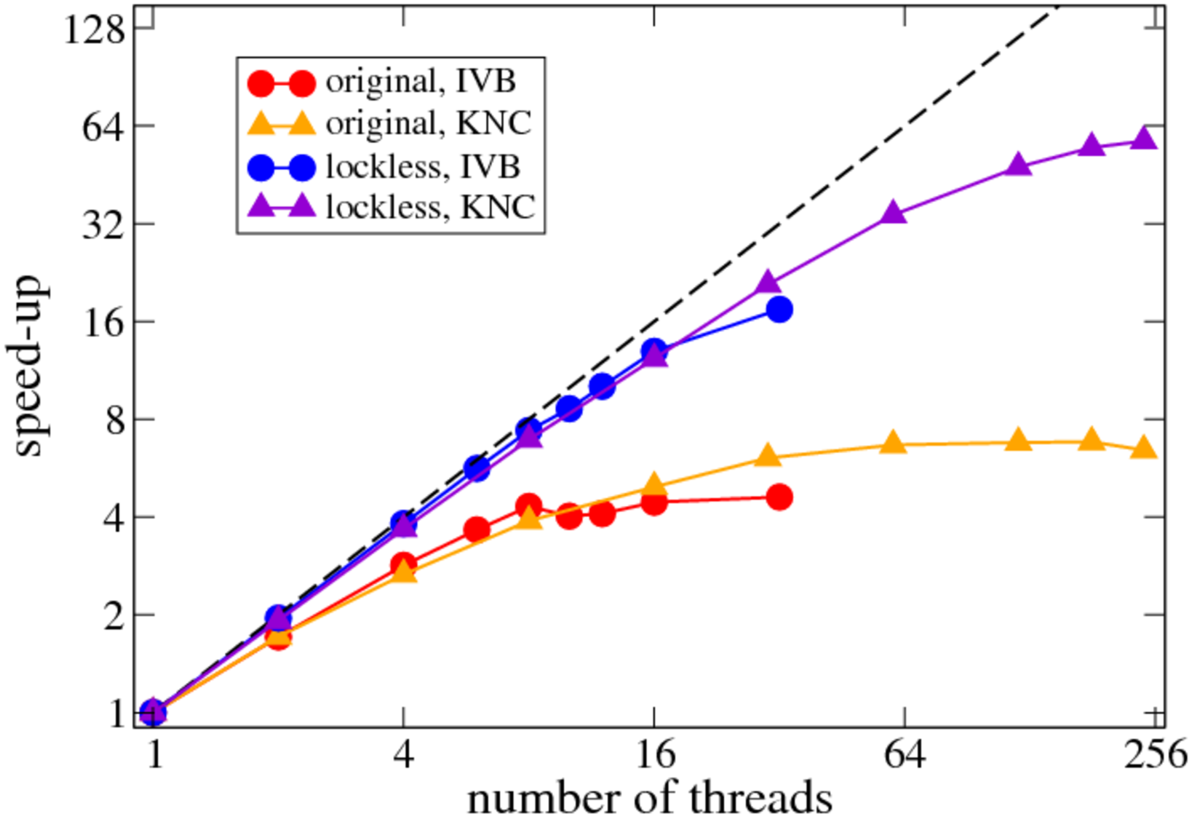
Viewing 1 of 1 selected stack(s)

100.0% (21.714s of 21.714s)

```
ipcc_kernels_openmp_ivb_novec!subfin…
ipcc_kernels_openmp_ivb_novec!subfin…
ipcc_kernels_openmp_ivb_novec!subfin…
ipcc_kernels_openmp_ivb_novec!subfin…
libiomp5.so![OpenMP dispatcher]+0x86…
libiomp5.so!__kmp_fork_call+0xff7 - km…
libiomp5.so![OpenMP fork]+0xd7 - kmp…
ipcc_kernels_openmp_ivb_novec!subfin…
ipcc_kernels_openmp_ivb_novec!main+…
libc.so.6!__libc_start_main+0xe5 - [unk…
ipcc_kernels_openmp_ivb_novec!_start…
```

- On IVB @ 8 threads
  - speed-up: 1.8x
  - parallel efficiency: 92%

- On KNC @ 60 threads
  - speed-up: 5.2x
  - parallel efficiency: 57%



Obstacles to efficient auto-vectorization

```
for(n = 0, n < neighboring_particles, n++ ){
    j = ngblist[n];

    if (particle n within smoothing_length){

       inlined_function1(…, &w);
       inlined_function2(…, &w);

       rho   += P_AoS[j].mass*w;
       vel_x += P_AoS[j].vel_x;
       …
       v2 += vel_x*vel_x + … vel_z*vel_z;
   }
```

for loop over neighbors

check for computation

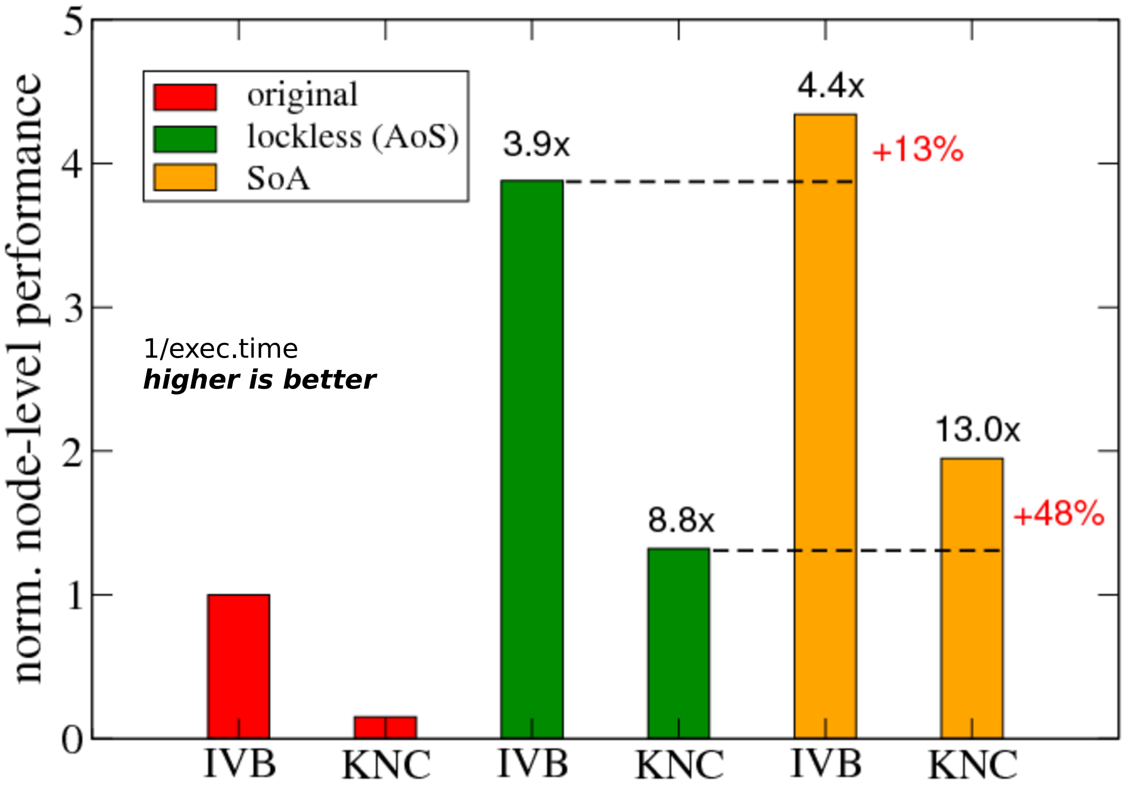
computing physics

Particles properties via **AoS** (cache unfriendly!)

$$\rho_i = \rho(\mathbf{r}_i) = \sum_j m_j W(|\mathbf{r}_i - \mathbf{r}_j|, h_j)$$

AoS to SoA: performance outcomes

- Gather+scatter overhead at most 1.8% of execution time. → intensive data-reuse
- Performance improvement:
- on IVB: 13%, on KNC: 48%
- Xeon/Xeon Phi performance ratio: from 0.15 to 0.45.
- The data structure is now vectorization-ready.



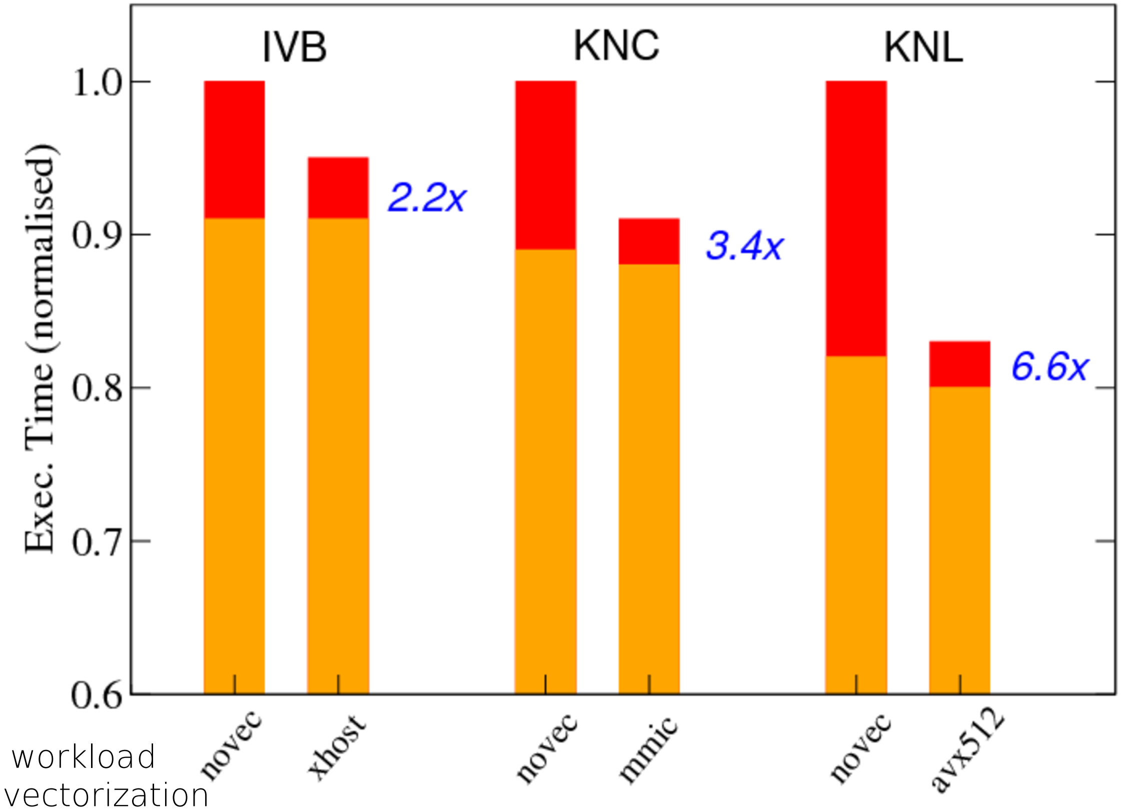
Vectorization: improvements from IVB to KNL

- Vectorization through localized masking (if-statement moved inside the inlined functions).

- Vector efficiency:

  perf. gain / vector length

  on IVB: 55%
  on KNC: 42%
  on KNL: 83%



- Yellow + red bar: kernel workload
- Red bar: target loop for vectorization

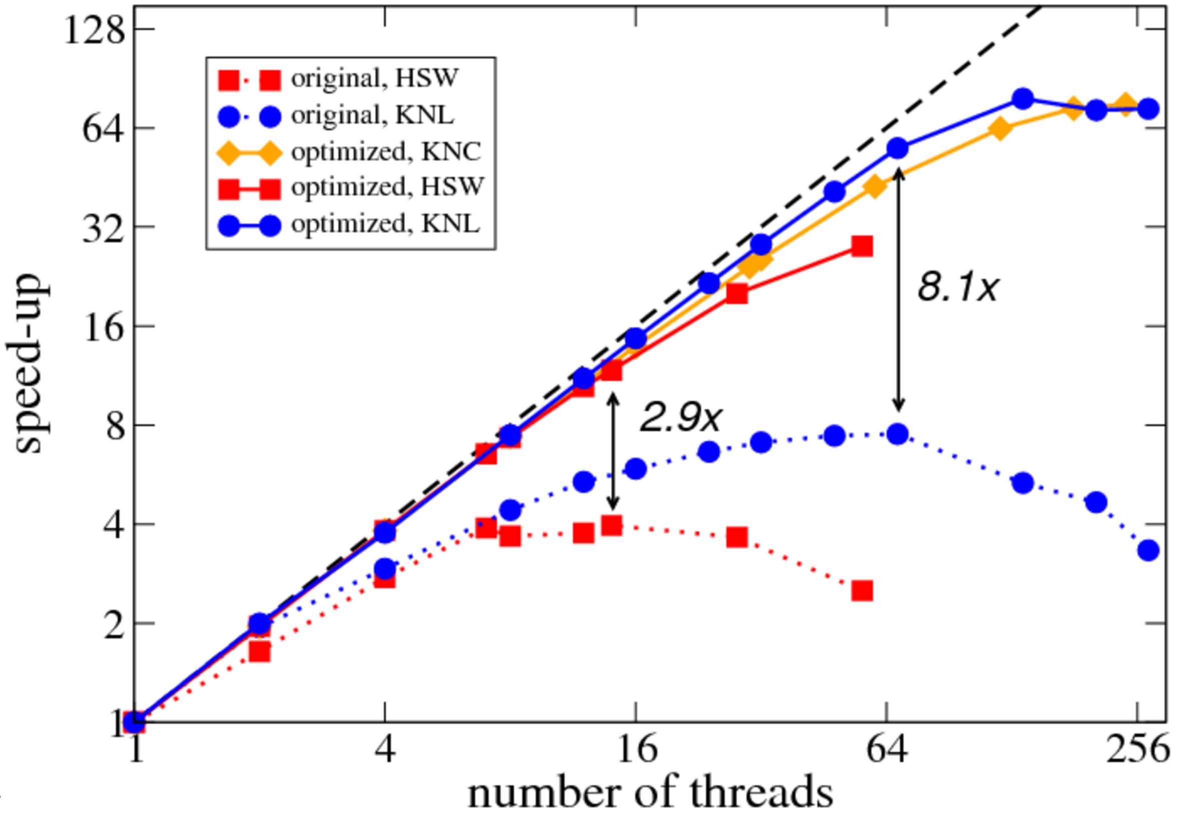
Node-level performance comparison between HSW, KNC *and* KNL

Features of the KNL tests:

- KMP Affinity: scatter;
  Memory mode: Flat;
  MCDRAM via numactl;
  Cluster mode: Quadrant.

Results:

- Our optimization improves the speed-up on all systems.
- Better threading scalability up to 136 threads on KNL.
- Hyperthreading performance is different between KNC and KNL

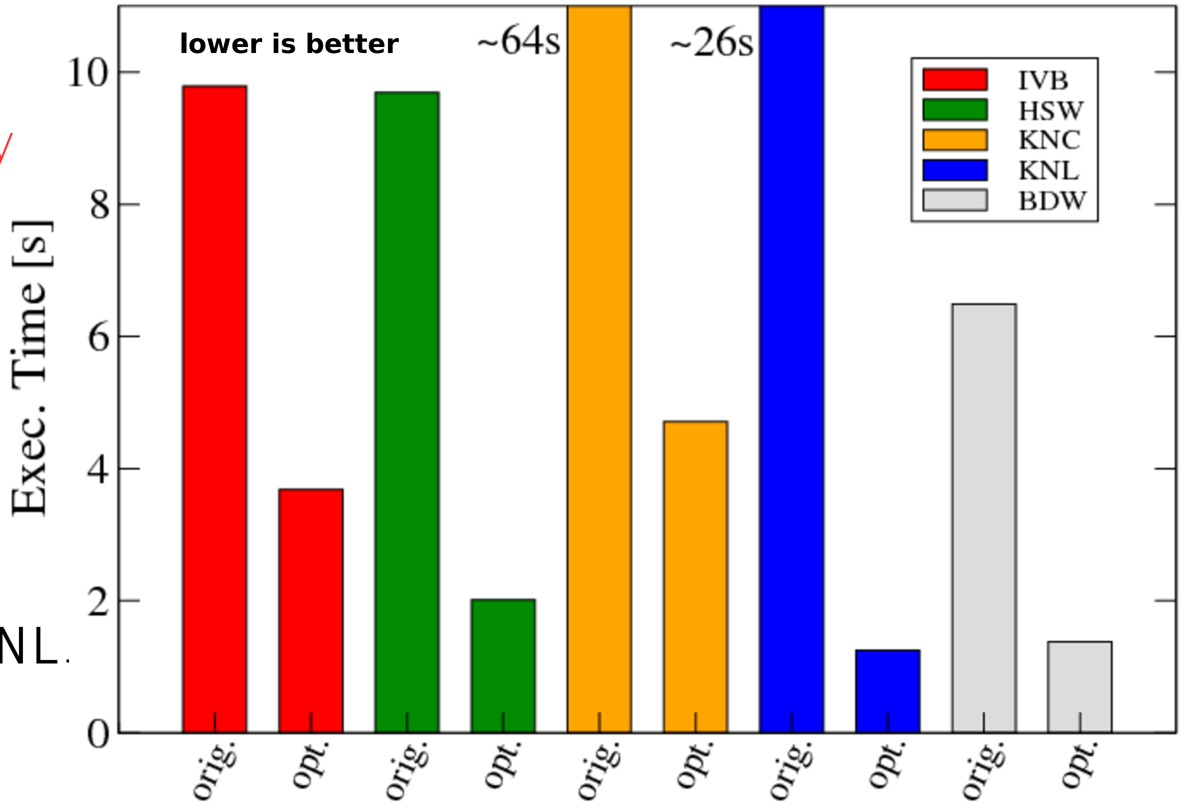


Performance comparison: first results including KNL and Broadwell

- Initial vs. optimized including all optimizations for subfind_density

- IVB, HSW, BDW: 1 socket w/o hyperthreading.
  KNC: 1 MIC, 240 threads.
  KNL: 1 node, 136 threads.

- Performance gain:
  - Xeon Phi: 13.7x KNC, 19.1x KNL.
  - Xeon: 2.6x IVB, 4.8x HSW, 4.7x BDW.



Code optimization on KNL: lessons learnt (so far...)

Optimization for KNL as a three-step process:

Step	Effort	Expected performance
Compilation "out of the box"	1 hour	Lower than Haswell (~ 1.5x)
Optimization without coding (use of AVX512, explore configuration, MCDRAM, MPI/OpenMP)	1 week	Up to 2x over previous step
Optimization with coding (this project and beyond)	1-3 months (IPCC: 2 years)	Up to the level of Broadwell

Freely adapted from Leijun Hu, Inspur @ ISC 2017

Some more KNL wisdom

- Quad-cache is a good starting point, quad-flat with allocation on MCDRAM is worth being tested, SNC modes are for very advanced developers.
- It is unlikely to gain performance with more than 2 threads/core.
- Vectorize whenever possible, use compiler reports and tools to exploit low-hanging fruits.
- Know where your data are located and how they move.
- If optimizations are portable, the effort pays off!

Summary and outlook

- Code modernization as the iterative process for improving the performance of an HPC application.
- Our IPCC example: P-Gadget3.

$$\left.\begin{array}{l}\text{Threading parallelism}\\ \text{Data layout}\\ \text{Vectorization}\end{array}\right\}\ \text{Key points of our work, guided by analysis tools.}$$

- This effort is (mostly) portable! Good performance found on new architectures (KNL and BDW) basically out-of-the-box.
- For KNL, architecture-specific features (MCDRAM, large vector registers and NUMA characteristics) are currently under investigation for different workloads.
- Investment on the future of well-established community applications, and crucial for the effective use of forthcoming HPC facilities.

Acknowledgements

- Research supported by the Intel® Parallel Computing Center program.
- Project coauthors: Nicolay J. Hammer (LRZ), Vasileios Karakasis (CSCS).
- P-Gadget3 developers: Klaus Dolag, Margarita Petkova, Antonio Ragagnin.
- Research collaborator at Technical University of Munich (TUM): Nikola Tchipev.
- TCEs at Intel: Georg Zitzlsberger, Heinrich Bockhorst.
- Thanks to the IXPUG community for useful discussion.
- Special thanks to Colfax Research for granting access to their computing facilities.

Back-up: removing lock contention

```
todo_partlist = partlist;

while(partlist.length){
  error=0;
  #pragma omp parallel for schedule(dynamic)
  for(auto p:todo_partlist){
    if(something_is_wrog) error=1;
    ngblist = find_neighbours(p);
    sort(ngblist);
    for(auto n:select(ngblist,K))
       compute_interaction(p,n);
  }

//...check for any error
  todo_particles = mark_for_recomputation(partlist);
}
```

creating a ***todo*** particle list

iterations over the ***todo*** list (*private ngblist*)

actual computation

No-checks for computation

Back-up: SoA implementation details

```
struct ParticleAoS
{
  float pos[3], vel[3], mass;
}
Particle_AoS *P_AoS;
P_AoS = malloc(N*sizeof(Particle_AoS);
```

```
struct ParticleSoA
{
  float *pos_x, … , *vel_x, …, mass;
}
Particle_SoA P_SoA;
P_SoA.pos_x = malloc(N*sizeof(float));
…
```

```
void gather_Pdata(struct Particle_SoA *dst, struct Particle_AoS *src, int N )
for(int i = 0, i < N, i++ ){
    dst -> pos_x[i] = src[i].pos[1]; dst -> pos_y[i] = src[i].pos[2]; …
}
```

```
…
rho   += P_AoS[j].mass*w;
vel_x += P_AoS[j].vel_x;
…
```

```
…
rho   += P_SoA.mass[j]*w;
vel_x += P_SoA.vel_x[j];
…
```